Gnu Awk - Part 14 (HPR Show 2816)

Redirection of input and output - part 1

Dave Morriss

Gnu Awk - Part 14 (HPR Show 2816)

Introduction

This is the fourteenth episode of the "Learning Awk" series which is being produced by b-yeezi and myself.

In this episode and the next I want to start looking at *redirection* within Awk programs. I had originally intended to cover the subject in one episode, but there is just too much.

So, in the first episode I will be starting with output redirection and then in the next episode will spend some time looking at the `getline` command used for *explicit input*, often with redirection.

Redirection of output

So far we have seen that when an `awk` script uses `print` or `printf` the output is written to the standard output (the screen in most cases). The *redirection* feature in `awk` allows output to be written elsewhere.

How this is achieved is described in the following sections.

Redirecting to a file

```
print items > output-file
printf format, items > output-file
```

Here, `'items'` denotes the items to be printed, `'format'` is the format expression for `'printf'`, `'output-file'` is an expression which is converted to a string and contains the name of the output file.

Here's a simple example. It uses the file of fruit data introduced in episode number 2. This data file is included with this show (awk14_fruit_data.txt):

```
$ awk 'NR > 1 {print $1 > "fruit_names"}' awk14_fruit_data.txt
$ cat fruit_names
```

```
apple
banana
strawberry
grape
apple
plum
kiwi
potato
pineapple
```

Here the script skips the first line of headers, then prints out the fruit name in field 1 to the file called `'fruit_names'`. Notice the file name is enclosed in quotes because it is a string.

The script will loop once per line of the input file executing the redirection each time. However the file contains all of the names in the same order as the input file. This is because of the following behaviour:

- The output file is erased before the first output is written to it.
- Subsequent writes to the same file do not erase it but append to it.

It is important to be aware that redirection in Awk is similar to but not the same as that in shell scripts.

What we have done here is not really different from running the following command where the shell deals with redirection:

```
$ awk 'NR > 1 {print $1}' awk14_fruit_data.txt > fruit_names
```

Here Awk is writing to the standard output stream and the shell is capturing this stream and redirecting it to a file. However, things get more complex if the requirement is to write to more than one file from a script.

The following downloadable script (awk14_ex1.awk) writes to a collection of output files:

```
$ cat awk14_ex1.awk
#!/usr/bin/awk -f

# Downloadable example 1 for GNU Awk Part 14

NR > 1 {
    colour = $2
    fname = "awk14_" colour "_fruit"
```

```
    printf "Writing %s to %s\n",$1,fname
    print $1 > fname
}
```

Running the script writes to files called `'awk14_brown_fruit'` and similar in the current directory:

```
$ ./awk14_ex1.awk awk14_fruit_data.txt
Writing apple to awk14_red_fruit
Writing banana to awk14_yellow_fruit
Writing strawberry to awk14_red_fruit
Writing grape to awk14_purple_fruit
Writing apple to awk14_green_fruit
Writing plum to awk14_purple_fruit
Writing kiwi to awk14_brown_fruit
Writing potato to awk14_brown_fruit
Writing pineapple to awk14_yellow_fruit
```

The script announces what it's doing, which is a little superfluous but helps to visualise what's going on.

Notice that since the output file names are generated dynamically and are liable to change between each line read from the input file the script is doing what was described earlier – creating them (or emptying them if they already exist) and then appending to them once open. All the files are closed when the script exits of course.

The files created are shown below and the contents of one displayed:

```
$ ls awk14_*_fruit
awk14_brown_fruit  awk14_green_fruit  awk14_purple_fruit
awk14_red_fruit  awk14_yellow_fruit

$ cat awk14_purple_fruit
grape
plum
```

Redirecting and appending to an existing file

The next type of redirection uses two greater than signs:

```
print items >> output-file
printf format, items >> output-file
```

In this case the output file is expected to exist already. If it does then its contents are not erased but are appended to. If the file does not exist then it is created and written to as before.

When redirecting to a file in a shell script it's common to see something like this:

```
echo "Script starting" > script.log
...
echo "Script ending" >> script.log
```

The use of `'>>'` in the second case is necessary because otherwise the file would have been cleared out before the message was written. Each redirection like this in Bash involves opening and closing the output file.

In an `awk` script on the other hand – as we have seen – the file is kept open by the script until it is closed on exit. There is a `'close'` command which will do this explicitly, and we will look at this shortly.

Redirecting to another program

This type of redirection uses a pipe symbol to send output to a string containing a command (or commands) for the shell.

```
print items | command
printf format, items | command
```

The following example shows the fruit names being written to a pair of commands in a shell pipeline:

```
$ awk 'NR > 1 {print $1 | "sort -u | nl"}' awk14_fruit_data.txt
     1  apple
     2  banana
     3  grape
     4  kiwi
     5  pineapple
     6  plum
     7  potato
     8  strawberry
```

The names are sorted using the `'sort'` command, requesting that the results be made unique (`'-u'`). The output from the sort is run through `'nl'` which numbers the lines.

As the awk script is run, a sub-process is executed with the two commands. The first name is then sent to this process, and this repeats with each successive name. The sub-process finishes when the script finishes.

In this case the 'sort' command will have accumulated all the names, then on the connection being terminated it will perform the sort and pass the results to 'nl'.

There is a 'close' command in awk which will close the redirection to the command(s) or to a file. The argument to 'close' needs to be the exact command(s) which define the process (or the exact file name). For this reason it's a good idea to store the commands or file name in an awk variable.

The following downloadable script (awk14_ex2.awk) shows the variable 'cmd' being used to hold the shell commands. The connection is closed to show how it would be done, though there is no actual need to do so here.

```
$ cat awk14_ex2.awk
#!/usr/bin/awk -f

# Downloadable example 2 for GNU Awk Part 14

BEGIN {
    cmd = "sort -u | nl"
}

NR > 1 {
    print $1 | cmd
}

END {
    close(cmd)
}
```

Running the script gives the same result as before:

```
$ ./awk14_ex2.awk awk14_fruit_data.txt
     1  apple
     2  banana
     3  grape
     4  kiwi
     5  pineapple
     6  plum
     7  potato
     8  strawberry
```

Here's a more *real world* example (at least it's real in my world). When I'm preparing an HPR show like this which involves a number of example scripts I need to run them for testing purposes. I have a main directory for HPR shows and a sub-directory per show. I like to make soft links to the examples in this sub-directory so I can run tests without hopping about between directories.

In general I make links in this way:

```
ln -s -f PathToExample BasenameOfExample
```

I wrote an Awk script to help me which takes path names as input and constructs shell commands which it pipes into `'sh'`.

The following downloadable script (awk14_ex3.awk) shows the process.

```
$ cat awk14_ex3.awk
#!/usr/bin/awk -f

# Downloadable example 3 for GNU Awk Part 14

{
    # Split the path up into components
    n = split($0,a,"/")
    if (n < 2) {
        print "Error in path",$0 > "/dev/stderr"
        next
    }

    # Build the shell command so we can show it
    cmd = sprintf("[ -e %s ] && ln -s -f %s %s",$0,$0,a[n])
    print ">> " cmd

    # Feed the command to the shell
    printf("%s\n",cmd) | "sh"
}

END {
    close("sh")
}
```

The script expects to be given one or more pathnames on standard input. It first takes the path and splits it up based on the `'/'` character. Since `'split'` returns the number of elements then that number will index the last element. We check that it's sensible before proceeding. Note that the error message generated by the `'if'` test is redirected to `'/dev/stderr'`. We'll be looking at this shortly.

We use `'sprintf'` to make the shell command. It first adds a test that the file path leads to a file, then if so the shell command uses the `'ln'` command to make a soft link. We use the `'-f'` option which forces the creation to proceed even if the link already exists. The first argument to `'ln'` is the path and the second the *basename* (last component) of the file path.

This command is printed for reference, then it is executed by printing to a process running `'sh'` (which will be the Bourne shell or similar by default).

Running the script can be achieved thus. We use `'printf'` as a simple way of adding a newline to each pathname. The paths come from a filename expansion which includes a question mark. Running it gives the following results:

```
$ printf "%s\n" Gnu_Awk__Part_14/hpr2816/awk14_ex?.awk | ./awk14_ex3.awk
>> [ -e Gnu_Awk__Part_14/hpr2816/awk14_ex1.awk ] && ln -s -f
Gnu_Awk__Part_14/hpr2816/awk14_ex1.awk awk14_ex1.awk
>> [ -e Gnu_Awk__Part_14/hpr2816/awk14_ex2.awk ] && ln -s -f
Gnu_Awk__Part_14/hpr2816/awk14_ex2.awk awk14_ex2.awk
>> [ -e Gnu_Awk__Part_14/hpr2816/awk14_ex3.awk ] && ln -s -f
Gnu_Awk__Part_14/hpr2816/awk14_ex3.awk awk14_ex3.awk
```

This is a script which I can use in all sorts of other contexts, though it probably needs some refinement to be completely foolproof.

Note that some caution is needed when writing shell commands in `awk` because of the potential pitfalls when using quotes. See the GNU Awk User's Guide section 10.2.9 for hints.

Redirecting to a *coprocess*

This type of redirection uses a pipe symbol and an ampersand to send output to a string containing a command (or commands) for the shell.

```
print items |& command
printf format, items |& command
```

This is an advanced feature which is a `gawk` extension. Unlike the previous redirection, which sends to a program, this form sends to a program **and** allows the program's output to be read back. That is why the command is referred to as a *coprocess*.

Since it is necessary to use our next main topic `'getline'` to achieve all of this we'll postpone discussing the subject until the next episode.

Redirecting to special files

There are three standard Unix channels that are known as *standard input*, *standard output*, and *standard error output* (or more commonly *standard error*). These are connected to keyboard and screen in the default case.

Normally a Unix program or script reads from *standard input* and writes to *standard output* and generates any error messages on *standard error*. There is a lot more to this than described here but this will suffice for the moment.

Gnu Awk can use three special file names to access these channels:

- `/dev/stdin`: standard input
- `/dev/stdout`: standard output
- `/dev/stderr`: standard error output

So, for example, a script can write explicitly to *standard error* with a command of the form:

```
print "Invalid number" > "/dev/stderr"
```

See the GNU Awk User's Guide section 5.7 on this subject for more details. There are also other special names available as described in the Guide in section 5.8.

Next episode

I will be continuing with the second half of this episode in a few weeks.

Links

- *GNU Awk User's Guide*
  - Redirecting output of `print` and `printf`
  - Special Files for Standard Preopened Data Streams
  - Special File names in gawk

- Previous shows in this series on HPR:
  - *"Gnu Awk - Part 1"* - episode 2114
  - *"Gnu Awk - Part 2"* - episode 2129

    - *“Gnu Awk - Part 3”* - episode 2143
    - *“Gnu Awk - Part 4”* - episode 2163
    - *“Gnu Awk - Part 5”* - episode 2184
    - *“Gnu Awk - Part 6”* - episode 2238
    - *“Gnu Awk - Part 7”* - episode 2330
    - *“Gnu Awk - Part 8”* - episode 2438
    - *“Gnu Awk - Part 9”* - episode 2476
    - *“Gnu Awk - Part 10”* - episode 2526
    - *“Gnu Awk - Part 11”* - episode 2554
    - *“Gnu Awk - Part 12”* - episode 2610
    - *“Gnu Awk - Part 13”* - episode 2804
- Resources:
    - ePub version of these notes
    - Examples: awk14_fruit_data.txt, awk14_ex1.awk, awk14_ex2.awk, awk14_ex3.awk